Learning Multiple Weights at a Time Generalizing Gradient Descent You don't learn to walk by following rules. You learn by doing, and by falling over. — RICHARD BRANSON 79 IN THIS CHAPTER Licensed to Asif Qamar 80 Chapter 5 I Learning Multiple Weights at a Time Gradient Descent Learning with Multiple Inputs Gradient Descent Also Works with Multiple Inputs In the last chapter, we learned how to use Gradient Descent to update a weight. In this chapter, we will more or less reveal how the same techniques can be used to update a network that contains multiple weights. Let's start by just jumping in the deep end, shall we? Th e following diagram lists out how a network with multiple inputs can learn! input data enters here (3 at a time) 1 An Empty Network With Multiple Inputs predictions come out here weights = [0.1, 0.2, -.1] def neural_network(input, weights): pred = w_sum(input,weights) return pred .1 .2 -.1 2 PREDICT+COMPARE: Making a Prediction and Calculating Error And Delta win loss #toes #fans win? toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.9, 0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] win_or_lose_binary = [1, 1, 0, 1] true = win_or_lose_binary[0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weights) error = (pred - true) ** 2 delta = pred - true .1 .2 -.1 8.5 65% 1.2 0.86 .020 error def w_sum(a,b): assert(len(a) == len(b)) output = 0 for i in range(a): output += (a[i] * b[i]) return output delta 0.14 prediction Licensed to Asif Qamar Gradient Descent Learning with Multiple Inputs 81 3 LEARN: Calculating Each "Weight Delta" and Putting It on Each Weight .1 .2 -.1 8.5 65% 1.2 0.86 .020 0.14 weight_deltas def ele_mul(number,vector): output = [0,0,0] assert(len(output) == len(vector)) for i in xrange(len(vector)): output[i] = number * vector[i] return output input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) error = (pred - true) ** 2 delta = pred - true weight_deltas = ele_mul(delta,weights) 1.19 .091 .168 There is actually nothing new in this diagram. Each weight delta is calculated by taking its output delta and multiplying it by its input. In this case, since our three weights share the same output node, they also share that node's delta. However, our weights have different weight deltas owing to their different input values. Notice further that we were able to re-use our ele_mul function from before as we are multiplying each value in weights by the same value delta. 8.5 * 0.14 = 1.19 = weight_deltas[0] 0.65 * 0.14 = 0.091 = weight_deltas[1] 1.2 * 0.14 = 0.168 = weight_deltas[2] 4 LEARN: Updating the Weights .0881 .191 -.102 win loss #toes #fans win? input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) error = (pred - true) ** 2 delta = pred - true weight_deltas = ele_mul(delta,weights) alpha = 0.01 for i in range(len(weights)): weights[i] -= alpha * weight_deltas[0] 0.1 - (1.19 * 0.01) = .0881 = weights[0] 0.2 - (.091 * 0.01) = .191 = weights[1] -0.1 - (.168 * 0.01) = -.102 = weights[2] Licensed to Asif Qamar 82 Chapter 5 I Learning Multiple Weights at a Time Gradient Descent with Multiple Inputs - Explained Simple to execute, fascinating to understand. When put side by side with our single-weight neural network, Gradient Descent with multiple inputs seems rather obvious in practice. However, the properties involved are quite fascinating and worthy of discussion. First, let's take a look at them side by side. 2 Multi Input: Making a Prediction and Calculating Error And Delta toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.9, 0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] win_or_lose_binary = [1, 1, 0, 1] true = win_or_lose_binary[0] # input corresponds to every entry # for the first game of the season input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) error = (pred - true) ** 2 delta = pred - true .1 .2 -.1 8.5 65% 1.2 0.86 .020 error delta 0.14 prediction 2 Single Input: Making a Prediction and Calculating Error and Delta 8.5 .023 -.15 number_of_toes = [8.5] win_or_lose_binary = [1] // (won!!!) input = number_of_toes[0] true = win_or_lose_binary[0] pred = neural_network(input,weight) error = (pred - true) ** 2 delta = pred - true delta error Indeed, up until the generation of "delta" on the output node, single input and multi-input Stochastic Gradient Descent is identical (other than the prediction diff erences we studied in Chapter 3). We make a prediction, calculate the error and delta in the identical ways. How ever, the following problem remains, when we only had one weight, we only had one input (one weight_delta to generate). Now

we have 3! How do we generate 3 weight_deltas? Licensed to Asif Qamar Gradient Descent with Multiple Inputs - Explained 83 How do we turn a single delta (on the node) into 3 weight_delta values? Let's remember what the defi nition and purpose of delta is vs weight_delta. Delta is a measure of "how much we want a node's value to be diff erent". In this case, we compute it by a direct subtraction between the node's value and what we wanted the node's value to be (pred - true). Positive delta indicates the node's value was too high, and negative that it was too low. .1 .2 -.1 8.5 65% 1.2 0.86 .020 0.14 prediction delta Consider this from the perspective of a input single weight, higlighted on the right. Th e delta says "Hey inputs! ... Yeah you 3!!! Next time, predict a little higher!". Th en, our single weight says, "hmm, if my input was 0, then my weight wouldn't have mattered and i wouldn't change a thing (stopping). If my input was negative, then I'd want to decrease my weight instead of increase (negative reversal). However, my input is positive and quite large, so I'm guessing that my personal prediction mattered a lot to the aggregated output, so I'm going to move my weight up a lot to compensate! (Scaling)". It then increases it's weight. So, what did those three properties/statements really say. Th ey all three (stopping, nega tive reversal, and scaling) made an observation of how the weight's role in the delta was aff ected by its input! Th us, each weight_delta is a sortof "input modifi ed" version of the delta. Bringing us back to our original question, how do we turn one (node) delta into three weight_delta values? Well, since each weight has a unique input and a shared delta, we simply use each respective weight's input multiplied by the delta to create each respective weight_delta. It's really quite simple. Let's see this process in action on the next page. weight_delta, on the other hand, is an estimate for the direction and amount we should move our weights to reduce our node delta, inferred by the derivative. How do we transform our delta into a weight_delta? We multiply delta by a weight's input. A measure of how much we want a node's value to be higher or lower to predict "perfectly" given the current training example. delta A derivative based estimate for the direction and amount we should move a weight to reduce our node_delta, accounting for scaling, negative reversal, and stopping. weight_delta Licensed to Asif Qamar 84 Chapter 5 I Learning Multiple Weights at a Time Below you can see the generation of weight_delta variables for the previous single-input architecture and for our new multi-input architecture. Perhaps the easiest way to see how similar they are is by reading the psudocode at the bottom of each section. Notice that the multi-weight version (bottom of the page), simply multiplies the delta (0.14) by every input to create the vari ous weight_deltas. It's really quite a simple process. 3 Multi: Calculating Each "Weight Delta" and Putting It on Each Weight .1 .2 -.1 8.5 65% 1.2 0.86 .020 0.14 weight_deltas def ele_mul(number,vector): output = [0,0,0] assert(len(output) == len(vector)) for i in xrange(len(vector)): output[i] = number * vector[i] return output input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) error = (pred - true) ** 2 delta = pred - true weight_deltas = ele_mul(delta,weights) 1.19 .091 .168 8.5 * 0.14 = 1.19 => weight_deltas[0] 0.65 * 0.14 = 0.091 => weight_deltas[1] 1.2 * 0.14 = 0.168 => weight_deltas[2] 3 Single Input: Calculating "Weight Delta" and Putting it on the Weight 8.5 .023 .1 -.15 number_of_toes = [8.5] win_or_lose_binary = [1] // (won!!!) input = number_of_toes[0] true = win_or_lose_binary[0] pred = neural_network(input,weight) error = (pred - true) ** 2 delta = pred - true weight_delta = input * delta weight delta -1.25 8.5 * -0.15 = -1.25 => weight_delta Licensed to Asif Qamar Gradient Descent with Multiple Inputs - Explained 85 4 Updating the Weights .0881 .191 -.102 win loss #toes #fans win? input = [toes[0],wlrec[0],nfans[0]] pred = neural_network(input,weight) error = (pred - true) ** 2 delta = pred - true weight_deltas = ele_mul(delta,weights) alpha = 0.01 for i in range(len(weights)): weights[i] -= alpha * weight_deltas[i] 0.1 - (1.19 * 0.01) = .0881 = weights[0] 0.2 - (.091 * 0.01) = .191 = weights[1] -0.1 - (.168 * 0.01) = -.102 = weights[2] .1125 4 Updating the Weight number_of_toes = [8.5] win_or_lose_binary = [1] // (won!!!) input = number_of_toes[0] true = win_or_lose_binary[0] pred = neural_network(input,weight) error = (pred - true) ** 2 delta = pred - true weight_delta = input *

delta alpha = 0.01 // fixed before training weight -= weight_delta * alpha We multiply our weight_delta by a small number "alpha" before using it to update our weight. This allows us to control how fast the network learns. If it learns too fast, it can update weights too aggressively and overshoot. More on this later. Note that the weight update made the same change (small increase) as Hot and Cold Learning new weight Finally, the last step of our process is also nearly identical to the single-input network. Once we have our weight_delta values, we simply multiply them by alpha and subtract them from our weights. It's literally the same process as before, repeated across multiple weights in stead of just a single one. Licensed to Asif Qamar 86 Chapter 5 I Learning Multiple Weights at a Time Let's Watch Several Steps of Learning .1 .2 -.1 8.5 65% 1.2 0.86 .020 -.14 weight_deltas -1.2 -.09 -.17 error weight error weight error weight def neural_network(input, weights): out = 0 for i in xrange(len(input)): out += (input[i] * weights[i]) return out def ele_mul(scalar, vector): out = [0,0,0] for i in xrange(len(out)): out[i] = vector[i] * scalar return out toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65, 0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] win_or_lose_binary = [1, 1, 0, 1] true = win_or_lose_binary[0] alpha = 0.01 weights = [0.1, 0.2, -.1] input = [toes[0],wlrec[0],nfans[0]] for iter in range(3): pred = neural_network(input,weights) error = (pred - true) ** 2 delta = pred - true weight_deltas=ele_mul(delta,input) print "Iteration:" + str(iter+1) print "Pred:" + str(pred) print "Error:" + str(error) print "Delta:" + str(delta) print "Weights:" + str(weights) print "Weight_Deltas:" print str(weight_deltas) print for i in range(len(weights)): weights[i]-=alpha*weight_deltas[i] a b c a b c 1 Iteration Notice that on the right, we can picture three individual error/weight curves, one for each weight. As before, the slopes of these curves (the dotted lines) are refl ected by the "weight_ delta" values. Furthermore, notice that (a) is steeper than the others. Why is the weight_delta steeper for (a) than the others if they share the same output delta and error measure? Well, (a) has an input value that is signifi cantly higher than the others. Th us, a higher derivative. Licensed to Asif Qamar Let's Watch Several Steps of Learning 87 .112 .201 -.098 8.5 65% 1.2 .964 .001 -.04 weight_deltas -.31 -.02 -.04 error weight error weight error weight a b c a b c 2 Iteration .115 .201 -.098 8.5 65% 1.2 .991 .000 -.01 weight_deltas -.08 -.01 -.01 error weight error weight error weight a b c a b c 3 Iteration A few additional takeaways: most of the learning (weight changing) was performed on the weight with the largest input (a), because the input changes the slope signifi cantly. Th is isn't necessarily advantagoues in all settings. Th ere is a sub-fi eld called "normalization" that helps encourage learning across all weights despite dataset characteristics such as this. In fact, this signifi cant diff erence in slope forced me to set the alpha to be lower than I wanted (0.01 instead of 0.1). Try setting alpha to 0.1. Do you see how (a) causes it to diverge? Licensed to Asif Qamar 88 Chapter 5 I Learning Multiple Weights at a Time Freezing One Weight - What Does It Do? .1 .2 -.1 8.5 65% 1.2 0.86 .020 -.14 weight_deltas -1.2 -.09 -.17 error weight error weight error weight def neural_network(input, weights): out = 0 for i in xrange(len(input)): out += (input[i] * weights[i]) return out def ele_mul(scalar, vector): out = [0,0,0] for i in xrange(len(out)): out[i] = vector[i] * scalar return out toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65, 0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] win_or_lose_binary = [1, 1, 0, 1] true = win_or_lose_binary[0] alpha = 0.3 weights = [0.1, 0.2, -.1] input = [toes[0],wlrec[0],nfans[0]] for iter in range(3): pred = neural_network(input,weights) error = (pred - true) ** 2 delta = pred - true weight_deltas=ele_mul(delta,input) weight_deltas[0] = 0 print "Iteration:" + str(iter+1) print "Pred:" + str(pred) print "Error:" + str(error) print "Delta:" + str(delta) print "Weights:" + str(weights) print "Weight_Deltas:" print str(weight_deltas) print for i in range(len(weights)): weights[i]-=alpha*weight_deltas[i] a b c a b c 1 Iteration Th is experiment is perhaps a bit advanced in terms of theory, but I think that it's a great exercise to understand how the weights aff ect each other. We're going to train again, except weight a won't ever be adjusted. We'll try to learn the training example using only weights b and c (weights[1] and weights[2]). Licensed to Asif Qamar Freezing One Weight - What Does It Do? 89 error weight error weight error weight a b c Iteration error weight error

weight error weight a b c Iteration 2 3 Perhaps you will be surprised to see that (a) still fi nds the bottom of the bowl? Why is this? Well, the curves are a measure of each individual weight relative to the global error. Th us, since the error is shared, when one weight fi nds the bottom of the bowl, all the weights fi nd the bottom of the bowl. Th is is actually an extremely im portant lesson. First of all, if we converged (reached error = 0) with (b) and (c) weights and then tried to train (a), (a) wouldn't move! Why? error = 0 which means weight_delt is 0! Th is reveals a potentailly damaging prop erty of neural networks. (a) might be a really powerful input with lots of predictive power, but if the network accidentally fi gures how how to predict accurately on the training data without it, then it will never learn to incorpo rate (a) into its prediction. Furthermore, notice "how" (a) fi nds the bottom of the bowl. Instead of the black dot moving, the curve seems to move to the left instead! What does this mean? Well, the black dot can only move horizontally if the weight is updated. Since the weight for (a) is frozen for this experiment, the dot must stay fi xed. However, the error clearly goes to 0. Th is tells us what the graphs really are. In truth, these are 2-d slices of a 4-di mensional shape. 3 of the dimensions are the weight values, and the 4th dimension is the error. Th is shape is called the "error plane" and, believe it or not, its curvature is deter miend by our training data! Why is it determined by our training data? Well, our error is determined by our training data. Any network can have any weight value, but the value of the "error" given any particular weight confi guration is 100% determined by data. We have already seen how the steepness of the "U" shape is aff ected by our input data (on several occasions). Truth be told, what we're really trying to do with our neural network is fi nd the lowest point on this big "error plane", where the lowest point refers to the "lowest error". Interesting eh? We're going to come back to this idea later, so just fi le it away for now. Licensed to Asif Qamar 90 Chapter 5 I Learning Multiple Weights at a Time Gradient Descent Learning with Multiple Outputs Neural Networks can also make multiple predictions using only a single input. Perhaps this one will seem a bit obvious. We calculate each delta in the same way, and then multiply them all by the same, single input. Th is becomes each weight's weight_delta. At this point, I hope it is clear that a rather simple mechanism (Stochastic Gradient Descent) is consistently used to perform learning across a wide variety of architectures. 1 An Empty Network With Multiple Outputs /* instead of predicting just whether the team won or lost, now we're also predicting whether they are happy/sad AND the percentage of the team that is hurt. We are making this prediction using only the current win/loss record */ weights = [0.3, 0.2, 0.9] def neural_network(input, weights): pred = ele_mul(input,weights) return pred input data enters here predictions come out here win loss win? sad? hurt? .3 .2 .9 2 PREDICT: Make a Prediction and Calculate Error and Delta .3 .2 .9 65% .195 .13 .585 wlrec = [0.9, 1.0, 1.0, 0.9] hurt = [0.1, 0.0, 0.0, 0.1] win = [1, 1, 0, 1] sad = [0.1, 0.0, 0.1, 0.2] input = wlrec[0] true = [hurt[0], win[0], sad[0]] pred = neural_network(input,weight) error = [0, 0, 0] delta = [0, 0, 0] for i in range(len(true)): error[i] = (pred[i] - true[i]) ** 2 delta = pred[i] - true[i] .009 .095 .757 .235 -.87 .485 Licensed to Asif Qamar Gradient Descent Learning with Multiple Outputs 91 3 COMPARE: Calculating Each "Weight Delta" and Putting It on Each Weight .2 65% .195 .13 .585 .009 .095 .757 .235 -.87 .485 .492 .153 .062 wlrec = [0.9, 1.0, 1.0, 0.9] hurt = [0.1, 0.0, 0.0, 0.1] win = [1, 1, 0, 1] sad = [0.1, 0.0, 0.1, 0.2] input = wlrec[0] true = [hurt[0], win[0], sad[0]] pred = neural_network(input,weight) error = [0, 0, 0] delta = [0, 0, 0] for i in range(len(true)): error[i] = (pred[i] - true[i]) ** 2 delta = pred[i] - true[i] weight_deltas = ele_mul(input,weights) def ele_mul(number,vector): output = [0,0,0] assert(len(output) == len(vector)) for i in xrange(len(vector)): output[i] = number * vector[i] return output weight_deltas As before, weight_deltas are computed by multiplying the input node value with the output node delta for each weight. In this case, our weight_deltas share the same input node and have unique output node (deltas). Note also that we are able to re-use our ele_mul function. 4 LEARN: Updating the Weights win loss win? sad? hurt? .29 .15 .89 input = wlrec[0] true = [hurt[0], win[0], sad[0]] pred = neural_network(input,weight) error = [0, 0, 0] delta = [0, 0, 0] for i in range(len(true)): error[i] =

(pred[i] - true[i]) ** 2 delta = pred[i] - true[i] weight_deltas = ele_mul(input,weights) alpha = 0.1 for i in range(len(weights)): weights[i] -= (weight_deltas[i] * alpha) Licensed to Asif Qamar 92 Chapter 5 I Learning Multiple Weights at a Time Gradient Descent with Multiple Inputs & Outputs Gradient Descent generalizes to arbitrarily large networks. 1 An Empty Network With Multiple Inputs & Outputs #toes %win #fans weights = [[0.1, 0.1, -0.3],#hurt? [0.1, 0.2, 0.0], #win? [0.0, 1.3, 0.1]]#sad? def neural_network(input, weights): pred = vect_mat_mul(input,weights) return pred .1 .2 .0 win loss #toes #fans win? sad? hurt? 2 inputs predictions 2 PREDICT: Make a Prediction and Calculate Error and Delta .1 .2 .0 2 inputs pred 8.5 65% 1.2 .555 .98 .965 toes = [8.5, 9.5, 9.9, 9.0] wlrec = [0.65,0.8, 0.8, 0.9] nfans = [1.2, 1.3, 0.5, 1.0] hurt = [0.1, 0.0, 0.0, 0.1] win = [1, 1, 0, 1] sad = [0.1, 0.0, 0.1, 0.2] alpha = 0.01 input = [toes[0],wlrec[0],nfans[0]] true = [hurt[0], win[0], sad[0]] pred = neural_network(input,weight) error = [0, 0, 0] delta = [0, 0, 0] for i in range(len(true)): error[i] = (pred[i] - true[i]) ** 2 delta = pred[i] - true[i] .207 -.02 .865 .96 .748 .455 errors Licensed to Asif Qamar Gradient Descent with Multiple Inputs & Outputs 93 3 .2 .0 2 inputs pred 8.5 65% 1.2 .555 .98 .965 .207 -.02 .865 .96 .748 .455 errors COMPARE: Calculating Each "Weight Delta" and Putting It on Each Weight .2 .562 .296 -.01 (weight deltas only shown for one input to save space) def outer_prod(vec_a, vec_b): out = zeros_matrix(len(a),len(b)) for i in range(len(a)): for j in range(len(b)): out[i][j] = vec_a[i]*vec_b[j] return out input = [toes[0],wlrec[0],nfans[0]] true = [hurt[0], win[0], sad[0]] pred = neural_network(input,weight) error = [0, 0, 0] delta = [0, 0, 0] for i in range(len(true)): error[i] = (pred[i] - true[i]) ** 2 delta = pred[i] - true[i] weight_deltas = outer_prod(input,delta) 4 LEARN: Updating the Weights inputs predictions input = [toes[0],wlrec[0],nfans[0]] true = [hurt[0], win[0], sad[0]] pred = neural_network(input,weight) error = [0, 0, 0] delta = [0, 0, 0] for i in range(len(true)): error[i] = (pred[i] - true[i]) ** 2 delta = pred[i] - true[i] weight_deltas = outer_prod(input,delta) for i in range(len(weights)): for j in range(len(weights[0])): weights[i][j] -= alpha * \ weight_deltas[i][j] .09 .2 .01 win loss #toes #fans win? sad? hurt? 2 Licensed to Asif Qamar 94 Chapter 5 I Learning Multiple Weights at a Time What do these weights learn? Each weight tries to reduce the error, but what do they learn in aggregate? Congratulations! Th is is the part of the book where we move onto our fi rst real world data set. As luck would have it, it's one with historical signifi cance! Our new dataset is called the MNIST dataset, which is a dataset comprised of digits that high school students and employees of the US Census bureau hand wrote some years ago. Th e interesting bit is that these handwritten digits are simply black and white images of people's handwriting. Accompanying each digit image is the actual number that they were writing (0-9). For the last few decades, people have been using this dataset to train neural networks to read human handwriting, and today, you're going to do the same! Each image is only 784 pixels (28 x 28). So, given that we have 784 pixels as input and 10 possible labels as output, you can imagine the shape of our neural network. So, now that each training example contains 784 values (one for each pixel), our neural network must have 784 input values. Pretty simple, eh! We just adjust the number of input nodes to refl ect how many data points are in each training example. Furthermore, we want to predict 10 probabilities, one for each digit. In this way, given an input drawing, our neural network will produce these 10 proba bilities, telling us which digit is most likely to be what was drawn. So, how do we confi gure our neural network to produce ten probabilties? Well, on the last page, we saw a diagram for a neural network that could take multiple inputs at a time and make multiple predictions based on that input. Th us, we should be able to simply modify this network to have the correct number of inputs and outputs for our new MNIST task. We'll just tweak it to have 784 inputs and 10 outputs. In the notebook entitled "", you'll see a script to pre-process the MNIST dataset and load the fi rst 1000 images and labels into two numpy matrices called images and labels. You may be wondering, "images are 2-dimensional... how do we load the (28 x 28) pixels into a fl at neural network?" For now, the answer is quite simple. We "fl at ten" the images into a vector of 1 x 784. So, we take the fi rst row of pixels and concatenate

them with the second row, and third row, and so on until we have one long list of pixels per image (784 pixels long in fact). Licensed to Asif Qamar What do these weights learn? 95 pix[0] pix[2] 1? 2? 0? inputs predictions pix[1] . . . pix[783] . . . 9? Th is picture on the left represents our new "MNIST Classifi cation" neural network. It most closely resembles the network we trained with "Multiple Inputs and Outputs" a few pages ago. Th e only diff er ence is the number of inputs and outputs, which has increased substantially. Th is network has 784 inputs (one for each pixel in a 28x28 image) and 10 outputs (one for each possible digit in the image). If this network was able to predict perfectly, it would take in an image's pixels (say a 2 like the one on the previous page), and predict a 1.0 in the correct out put position (the third one) and a 0 everywhere else). If it was able to do this correctly for all of the images in our dataset, it would have no error. 0.0 0.98 0.03 0.98 0.01 inputs predictions 0.0 . . . 0.95 . . . 0.15 Highest Prediction! Thus the network thinks that this image is a "2" Small Errors: This network thinks it kindof looks like a 9 (but only a bit) Over the course of training, the network will adjust the weights between the "input" and "prediction" nodes so that the error falls toward 0 in training. However, what does this actually do? What does it mean to modify a bunch of weights to learn a pattern in aggregate? Licensed to Asif Qamar 96 Chapter 5 I Learning Multiple Weights at a Time Visualizing Weight Values Each weight tries to reduce the error, but what do they learn in aggregate? pix[0] pix[2] 1? 2? 0? inputs predictions pix[1] . . . pix[783] . . . 9? pix[0] pix[2] 1? 2? 0? inputs predictions pix[1] . . . pix[783] . . . 9? Perhaps an interesting and intuitive practice in neural network research (particularly for image classifi ers) is to visualize the weights as if they were an image. If you look at the diagram on the right, you will see why. Each output node has a weight coming from every pixel. For example, our "2?" node has 784 input weights, each mapping the relationship between a pixel and the number "2". What is this relation ship? Well, if the weight is high, it means that the model believes there's a high degree of correlation between that pixel and the number 2. If the number is very low (neg ative), then the network believes there is a very low correlation (perhaps even negative correlation) between that pixel and the number two. Th us, if we take our weights and print them out into an image that's the same shape as our input dataset images, we can "see" which pixels have the highest correlation with a particular output node. As you can see above, there is a very vague "2" and "1" in our two images, which were created using the weights for "2" and "1" respectively. Th e "bright" areas are high weights, and the dark areas are negative weights. Th e neutral color (red if you're reading this in color) represents 0s in the weight matrix. Th is describes that our network generally knows the shape of a 2 and of a 1. Why does it turn out this way? Well, this takes us back to our lesson on "dot prod ucts". Let's have a quick review, shall we? Licensed to Asif Qamar Visualizing Dot Products (weighted sums) 97 Recall how dot produts work. Th ey take two vectors, multiply them together (ele mentwise), and then sum over the output. So, in the example below: First, you would multiply each element in a and b by each other, in this case creat ing a vector of 0s. Th e sum of this vector is also zero. Why? Well, the vectors had nothing in common. However, dot products between c and d return higher scores, because there is over lap in the columns that have positive values. Furthermore, performing dot products between two identical vectors tend to result in higher scores as well. Th e takeaway? A dot product is a loose measurement of similarity between two vectors. What does this mean for our weights and inputs? Well, if our weight vector is simi lar to our input vector for "2", then it's going to output a high score because the two vectors are similar! Inversely, if our weights vector is NOT similar to our input vector for 2, then it's going to output a low score. You can see this in action below! Why is the top score (0.98) higher than the lower one (0.01)? a = [0, 1, 0, 1] b = [1, 0, 1, 0] [0, 0, 0, 0] -> 0 b = [1, 0, 1, 0] c = [0, 1, 1, 0] c = [0, 1, 1, 0] d = [.5, 0,.5, 0] Visualizing Dot Products (weighted sums) Each weight tries to reduce the error, but what do they learn in aggregate? 0.98 0.01 inputs predictions (dot) weights score (equals) Licensed to Asif Qamar 98 Chapter 5 I Learning Multiple Weights at a Time Conclusion Gradient Descent is a General Learning Algorithm Perhaps the

most important subtext of this chapter is that Gradient Descent is a very flexible learning algorithm. If you combine weights together in a way that allows you to calculate an error function and a delta, gradient descent can show you how to move your weights to reduce your error. We will spend the rest of this book exploring different types of weight combinations and error functions for which Gradient Descent is useful. The next chapter is no exception